Experiences on the KSR1*Computer

Erik Boman
Stanford University

January 15, 1993

1 Introduction

This report is a summary of the experiences a novice user of the KSR-1 (Kendall Square Research) had trying to run the NPB (NAS Parallel Benchmarks) on this computer. The object of this project was to try to estimate the amount of human effort required to get a standard FORTRAN 77 program running efficiently on the KSR1. This implicitly implies an evaluation of the KSR compiling system and the entire programming environment.

We first give a brief description of the KSR1 computer and system software. In the next sections, we present our practical experiences with compiling, running, and modifying the code and give some performance results. Finally, we try to draw some conclusions.

2 The KSR1 Computer

2.1 Hardware

The KSR1 is a shared-memory, multiprocessor computer. It claims to be the first shared-memory machine which has the scalability of highly parallel systems. This is mainly due to its memory system, ALLCACHE.

Logically, the KSR1 consists of a collection of processors connected to a shared memory with 2^{40} bytes (virtual) address space. Physically, the processors are in a ring and the memory is distributed among the processors. Each processor has a local cache of 32 MB (all memory is cache). The ALLCACHE system provides a search engine that maps virtual memory

*KSR and KSR1 are registered trademarks of the Kendall Square Research Corporation.

addresses to local processors and dynamically moves data during execution. The actual physical location of any memory address is transparent to the user, and will vary with time. Therefore the user can think of the KSR1 as a shared memory machine although the memory is physically distributed among the processors. Since all memory is cache and the search engine is very fast, the memory access time does not vary as much as on a conventional distributed-memory system. But the access time will depend on where the data is located, so data locality is an important issue.

The KSR1 we used for this project was located at Oak Ridge and had 64 processors.

2.2 Software

The KSR1 runs under the operating system KSR-OS, which is an OSF/1 based UNIX variant. It has both a C and a Fortran compiler. We will only consider the Fortran programming environment. The components of this environment are:

- KSR Fortran - the Fortran compiler.
- KSR KAP - a source-to-source preprocessor to assist parallelizing the code.
- Presto - a run-time library that manages the execution of parallel constructs

The KSR Fortran language is an extended version of Fortran 77. It includes some parts of Fortran 90, e.g. dynamic arrays (allocate) and the use of array sections. But it is far from fulfilling the requirements of the Fortran 90 or the HPF standard. KSR Fortran relies heavily on the use of KSR-specific compiler directives for managing data layout and for parallel execution.

2.3 Levels of parallel programming

There are two main levels for parallel programming in Fortran on the KSR:

1. High level: Indirectly through KSR Fortran statements and directives.
2. Low level: By explicit use of pthreads.

A pthread is a single sequential flow of control within a process. A task begins with one pthread and creates others to perform work in parallel. KSR Fortran provides an interface to the pthreads very similar to the C interface defined in the POSIX standard.

However, since an important feature of the KSR KAP and the KSR Fortran is the high-level, easy-to-use parallel directives, we decided to concentrate on this approach in our work. This will also be the natural choice for programmers with limited time to rewrite Fortran 77 code. But it is important to note that although the high-level interface is easier to use, the low-level interface gives the programmer more explicit control over the processors/processes and the data management.

3 Running the NPB on the KSR1

3.1 Trying the simple way first

Our initial plan was to take the NAS Parallel Benchmark codes ([**?**]) and run then on the KSR1: First as sequential programs on a single processor, and then the same code compiled as a parallel program running on all processors. By not changing the source code, we would then effectively test the KAP preproceocessor's ability to parallelize sequential Fortran 77 code.

Such an automatic parallel compilation was done by the command

```
f77 -r8 -kap -para embar.f -lpresto
```

(Note that the `-r8` option is necessary to force double precision numbers to be 8 bytes instead of the default 16 bytes).

The result was discouraging. Most of the 8 benchmark programs ran insignificantly faster on p ($1 < p \leq 32$) processors than on one processor, simply because most parts of the programs were not parallelized. Looking at the log-files, it was clear that the main reason for this was that many loops either were not possible to run in parallel as they were written, or that KSR KAP believed it was not safe to do so because of data dependency. The result was really not very surprising, the experiment mainly shows that there is still a long way to go before one can give an old Fortran 77 program to a parallel computer and get it to run with a high degree of parallelism (it is indeed doubtful if this will ever happen).

The next step was then to follow the procedure a scientist or programmer would have to do if she wants to utilize more than one processor: Make changes in the code, add compiler directives etc. Since we had limited time for this project, the simplest benchmark program was chosen for this purpose: The embarrassingly parallel benchmark (EP).

3.2 The EP benchmark

The EP benchmark is a kernel that could be used in a Monte Carlo type procedure. $2n$ pseudorandom floating point numbers are generated and some computations which depend only on pairs of neighboring elements are performed. The data dependency is minimal, hence the problem is called embarrassingly parallel. It should be mentioned that the sought result is a table of 10 integer values, which are counts of how many pseudorandom pairs satisfy some specified criteria.

The compute intensive part of the program is a loop over segments of the $2n$ numbers. For each iteration, a batch of pseudorandom numbers is first generated and then analyzed. The numbers are generated in batches instead of one-by-one because this is normally faster, even if this part of the code is not easily vectorizable. The obvious strategy on a parallel computer is to give each processor such a segment of numbers to work on. These computations are completely independent, except that the answer is a combination of all the answers to the subproblems.

The EP problem is somewhat untypical for a general numerical program. In many real-life cases one does not know how to distribute work and data among processors, and one is more than happy to think in terms of shared-memory and leave this task to the compiling and run-time system. But in our case, we have the opposite situation. We know how to distribute work and data We would like to think in terms of local (distributed) memory. It turned out that the greatest problem was, in fact, to get the KSR1 to do what we wanted it to do.

The most important high-level parallel directives in KSR Fortran are the *tiling* constructs. These are used for splitting up loops into independent tasks that can be executed in parallel. The word *tile* is used because the index space is subdivided into tiles that are assigned to different pthreads (processors). Various options control the tiling strategy, the tile size, and related parameters.

3.2.1 Tiling the main loop

First we had to parallelize the main loop (DO 150). KSR KAP could not do this on its own since the loop contains a subroutine call with an array

```
C
C   Each instance of this loop may be performed independently.
C
      DO 150 K = 1, NN
        KK = K - 1
        T1 = S
        T2 = AN
C
C   Find starting seed T1 for this KK.
C
        DO 120 I = 1, 100
          IK = KK / 2
          IF (2 * IK .NE. KK) T3 = RANDLC (T1, T2)
          IF (IK .EQ. 0) GOTO 130
          T3 = RANDLC (T2, T2)
          KK = IK
 120    CONTINUE
C
C   Compute uniform pseudorandom numbers.
C
 130    CALL VRANLC (2 * NK, T1, A, X)
C
C   On a single processor system, the 120 loop and line 130 can be replaced
C   by the following single line.
C
C        CALL VRANLC (2 * NK, TT, A, X)
C
C   Compute Gaussian deviates by acceptance-rejection method and tally counts
C   in concentric square annuli.   This loop is not vectorizable.
C
        DO 140 I = 1, NK
          X1 = 2.D0 * X(2*I-1) - 1.D0
          X2 = 2.D0 * X(2*I) - 1.D0
          T1 = X1 ** 2 + X2 ** 2
          IF (T1 .LE. 1.D0) THEN
            GC = GC + 1.D0
            T2 = SQRT (-2.D0 * LOG (T1) / T1)
            T3 = ABS (X1 * T2)
            T4 = ABS (X2 * T2)
            L = MAX (T3, T4)
            Q(L) = Q(L) + 1.D0
          ENDIF
 140    CONTINUE
C
 150  CONTINUE
```

Figure 1: The main loop in `embar.f`

parameter. This creates potential write-conflicts. Therefore we have to tell that it is safe to do so by adding directives of the type

```
assert ignore calls
assert ignore any dependency
```

But this was not sufficient to make KAP tile the loop, because there were still data dependencies. We then tried manual tiling, which gives the programmer more control. But then we also had to declare which variables should be "shared" and which should be "private". This requires the programmer to carefully check the indices and variables in that loop, as it is easy to make a small error which may lead to completely wrong results.

Example of tiling directive used:

```
C*KSR* USER TILE(K, TILESIZE=K:ISIZ,
C*KSR*& PRIVATE=(KK,IK,T1,T2,T3,T4,L,X1,X2,I))
```

3.2.2 Data layout

We want different subprocesses (threads) to work on separate segments of the total interval of numbers. There are two obvious ways to handle this: Either to declare one large matrix and try to adjust the tiling directives such that each processor works on a local portion of it, or to declare local arrays (as local variables on each processor). Since the KSR emphasizes the shared-memory model it does not have any support for defining local variables. The closest feature we have been able to find is the *private* attribute, but this can not be used on arrays. Consequently, we had to declare one single, large array and hope that the compiling and run-time system would find out how to store the data in a good way.

3.2.3 Combining the partial results

The problem is classic: We have p processors that are each computing a part of the solution. The question is how to combine the partial answers. There are two basic ways to do this is:

- write to a shared variable. This requires synchronization and lock/unlock while updating the solution array.
- let each processor (pthread) have its own local array variable and combine the partial results in the end.

#proc.	Time	Speedup
1	195.1	1.00
2	172.8	1.13
4	127.2	1.54
8	75.2	2.59
16	55.0	3.55
32	45.8	4.26

Figure 2: Execution time and speedup for EP on the KSR1

We tried both approaches. The latter turned out to be the faster. In general, having many pthreads read from the same variable is fast on the KSR (since then copies of the variable may exist in several caches), but writing to memory is (much) slower.

3.2.4 Timings

We ran the best parallel version of the EP we developed on the KSR1, varying the number of processors from 1 to 32**??**. All times are given in seconds. The speedup is here defined as $\frac{t_p}{t_1}$, with t_p = parallel execution time with p processors. Note that the program was run with the small ("workstation") problem dimension $n = 2^{24}$ instead of the full scale problem $n = 2^{28}$, which would have taken approximately 16 times as long time.

For comparison, we ran the original sequential Fortran 77 code on 1 and 32 processors and got the times 195.0 and 196.0, respectively. So at least this matches well with the time spent running the parallel code on one processor. The problem with the timings is that we expected near linear speedup. Instead we observe that the speedup behaves approximately like $\log_2 p$ with p processes. We have been unable to explain why we did not get better speedup. The last step combining the partial results will need at least $\log_2 p$ steps, but the main calculations should be perfectly partioned among the processors and thus reduce the run time by a factor $\approx p$.

Clearly, these poor results are not optimal on the KSR1. In [**?**] it is reported the a 32 processor KSR1 solved the EP with $n = 2^{28}$ in 69.8 seconds. This would give a speedup of 44.7 compared to our timings. This is not unreasonable since linear speedup would give a factor 32 and additional optimization has surely been done on this version. So the conclusion is that we were unable to attain (near) linear speedup because of some unknown flaw in our implementation.

4 General remarks on programming on the KSR1

4.1 Documentation

We had the following documentation available:

- KSR Parallel Programming.
  Contains an introduction to the KSR1, describes the parallel support for Fortran Programs (e.g. tiling strategies), the use of pthreads, and data management.

- KSR Fortran Programming.
  Contains an introduction to the KSR Fortran programming environment, and detailed descriptions of the KSR Fortran language, the compiler directives, and the KAP preprocessor. Includes also a chapter on pthreads.

The author found the documentation to be overall good and extensive. But the manuals were too long and detailed for the first-time user, who would need a shorter hands-on introduction.

But there are some details to critisize. Most important: there is practically no information on how one should control the number of processors one uses during execution! This caused some confusion, and the KSR OS command *allocate_cells* was only found by searching the hard way. The PL_NUM_THREADS variable is briefly described in the documentation, but its relation to *allocate_cells* is not mentioned.

4.2 Reliability and robustness

Our impression of the KSR1 during the short project period (Nov 92 - Jan 93) was that the machine still is not sufficiently stable for industrial use. On several occasions the computer would suddenly just "hang" and needed a reboot to get working again. We also had problems with parallel programs that never finished execution for some unknown reason. Note that this was not necessarily a hardware or software error, but could be just incompetent programming.

5 Summary

The KSR has been promoted as the parallel computer that is easy to use. Although it is possible to run a program in parallel without any adaption, our experiment suggests that naive usage gives quite poor performance. To achieve high performance one has to use low-level parallel programming and do manual tuning specific for the KSR, just as for most parallel computers.

The ALLCACHE system has undoubtably some strong sides, e.g. it is very useful for dynamical jobs where the programmer does not know a good load-balancing or memory layout strategy in advance. But on the other hand, if this is known to the programmer, it may be a non-trivial task to explain this to the compiling system! Undoubtably, this can be done. It is certainly possible to get near linearly speedup if we only "do things right". But this turned out to be harder than expected.

We emphasize that these conclusions are drawn based a very limited test project and do not in any way pretend to be a complete evalution of the KSR1.

References

[1] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. NAS parallel benchmark results. *IEEE J. Parallel and Distributed Technology*, 1(1):43 – 51, 1993.

[2] D. H. Bailey, J. Barton, T. Lasinski, and H. (editors) Simon. The NAS parallel benchmarks. Technical Report RNR-91-02, NASA Ames Research Center, Moffett Field, CA 94035, January 1991.